Using TAME to Prove Invariants of Automata Models: Two Case Studies [*]

Myla Archer
Naval Research Laboratory
Code 5546
Washington, DC 20375
archer@itd.nrl.navy.mil

Constance Heitmeyer
Naval Research Laboratory
Code 5546
Washington, DC 20375
heitmeyer@itd.nrl.navy.mil

Elvinia Riccobene
Dipartimento di Matematica e
Informatica
Università di Catania
Viale A. Doria 6, I-95125
Catania, Italy
riccobene@dmi.unict.it

ABSTRACT

TAME is a special-purpose interface to PVS designed to support developers of software systems in proving properties of automata models. One of TAME's major goals is to allow a software developer who has basic knowledge of standard logic, and can do hand proofs, to use PVS to represent and to prove properties about an automaton model without first becoming a PVS expert. A second goal is for a human to be able to read and understand the content of saved TAME proofs without running them through the PVS proof checker. A third goal is to make proving properties of automata with TAME less costly in human time than proving such properties using PVS directly. Recent work by Romijn and Devillers et al., based on the I/O automata model, has provided the basis for two case studies on how well TAME achieves these goals. Romijn specified the RPC-Memory Problem and its solution, while Devillers et al. specified a tree identify protocol. Hand proofs of specification properties were provided by the authors. In addition, Devillers et al. used PVS directly to mechanize the specifications and proofs of the tree identify protocol. In one case study, the third author, a new TAME user with no previous PVS experience, used TAME to create PVS specifications of the I/O automata presented by Romijn and Devillers et al. and to check the hand proofs of invariant properties. The PVS specifications and proofs of Devillers et al. provide the basis for the other case study, which compares the TAME approach to an alternate approach which uses PVS directly.

Keywords

Software engineering, software requirements analysis, formal methods, verification, theorem proving.

1. INTRODUCTION

Several authors [25, 8, 9, 7] have found that the PVS specification language and similar strongly typed, higher-order logic languages are well suited to the formalization of system specifications. All report that appropriately structured PVS specifications can be understood by practitioners, such as design engineers and requirements analysts.

Yet, a number of barriers exist to more widespread industrial use of formal techniques such as PVS. Although Miller notes in [25] that engineers at Collins Aviation learned to use PVS, the authors of each of [25, 8, 9, 10] concede that in general practitioners themselves may be unwilling or unable to create formal specifications or to perform analysis of the specifications using the PVS proof checker. Further, Butler et al. [8] observe that one barrier to transferring formal methods technology to industry is that:

> Training the industrial experts to use the formal techniques, especially to develop skill in verification, [is costly].

The high cost of training industrial experts is not the sole barrier to more general use of PVS (or other theorem proving systems) to formalize and to prove properties of specifications. For a given system, a user must discover an appropriate representation in the language of the theorem prover. Formal proofs of system properties will be influenced by the details of the representation, the particular reasoning steps available in the theorem prover, and the manner in which the properties are formalized. The effort required therefore is substantial even for those who have mastered the use of the theorem prover. Moreover, direct use of any general theorem prover can encourage users to adapt both specifications and proofs to the needs of the prover rather than to the conventions associated with the problem domain. As a result, practitioners may find it difficult to relate the formalization of their system and formal proofs of its properties to their understanding of the system behavior and their beliefs as to why it has certain properties. We agree with Butler et al. [8] when they state that:

> [T]he formal methods researchers must be willing to adapt their methods to the problem domain rather than fight to change the existing methodologies to conform to their needs.

TAME (Timed Automata Modeling Environment) [3, 5, 6, 2] is a specialized interface to PVS that is intended to re-

Report Documentation Page

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
2000		**00-00-2000 to 00-00-2000**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Using TAME to Prove Invariants of Automata Models: Two Case Studies	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Naval Research Laboratory,Code 5546,4555 Overlook Avenue, SW,Washington,DC,20375	

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE		**12**	
unclassified	**unclassified**	**unclassified**			

move, or at least reduce, the barriers to more general use of PVS in verifying automata models. It supports the creation of PVS descriptions of three different automata models: Lynch-Vaandrager (LV) timed automata [23], I/O automata [21], and the automata model that underlies SCR specifications [16]. Further, TAME supports formulating properties of these automata in standard logic and proving the properties using reasoning "natural" to humans. A major goal of TAME is to improve on the direct support that PVS provides for specifying, and proving properties of, automata. To make PVS specifications of automata easy to create from automaton descriptions, TAME provides templates for specifying automata. To make PVS proofs of properties of automata easy to create, TAME provides PVS strategies which support the kinds of proof steps normally found in hand proofs of invariant properties of automata. In particular, TAME is intended to support straightforward construction of a mechanized proof of an invariant property from a high level hand proof that expresses an application expert's explanation of why the property should hold.

In recent work by Romijn [28, 27] and Devillers et al. [12, 11], example systems to be implemented in software were specified and verified using the I/O automata model. This work, together with the availability of a volunteer TAME user with no previous PVS experience (the third author), gave us an opportunity to evaluate the extent to which TAME has achieved its goals. Both [28] and [12] provide descriptions of I/O automata and formulations of their properties. References [27] and [11] provide hand proofs of most of the properties in [28] and [12] in varying degrees of detail. Thus, the third author had several example specifications and proofs to which she could apply TAME. In addition, reference [12] describes the PVS specification and proofs for one application, an I/O automaton called *TIP*, with a pointer to the actual specification and proofs on the web. This gave us the opportunity to compare the use of TAME with the more direct use of PVS on one particular problem.

Section 2 of this paper provides a high level description of TAME, its templates, and its strategies. Section 3 then compares the PVS approach to specification and proof for *TIP* [12] to the TAME approach to the same problem. (Although this case study was done after the study of *TIP* by the third author discussed in Section 4, we describe it first because it clarifies the difference between using PVS through TAME and using PVS directly.) Section 4 describes the experience of the third author in applying TAME to the examples of references [28, 27, 12, 11], with particular attention to 1) the time and effort required and 2) the adequacy of the TAME proof steps for mechanizing the proofs of invariants. Section 5 discusses the results of Sections 3 and 4 and describes some improvements in TAME that resulted from the case study in Section 4. Finally, Section 6 describes related work, and Section 7 presents our conclusions.

2. TAME

For software developers in industry, model checking is often viewed as more practical than theorem proving for establishing properties of software systems. While clearly an important technique for developing correct software systems, model checking does not solve all of a software developer's problems. For example, although model checking is often described as automatic and therefore requiring less exper-

tise from the user, the user, due to the state explosion problem, must typically model check an abstraction of a given system rather than the full system. Finding the appropriate abstraction often requires user ingenuity and creativity. Even when abstraction is used, state explosion can prevent a model checker from running to completion, and thus from establishing the correctness of a property. Moreover, in specifications involving parameters, model checking alone can only check correctness for specific (usually small), rather than arbitrary, values of the parameters. The protocols from [28] and [12] involve both parameters and another feature— complex data types—problematic for a model checker. Thus, for the verification of these and similar examples, theorem proving is necessary.

TAME is intended to reduce the human effort associated with mechanical theorem proving using PVS. To achieve this, TAME provides an interface to PVS [3, 5, 6, 2]. This interface consists of a set of templates for specifying automata, a set of standard theories, and a set of standard PVS strategies. Below, we provide an overview of the templates, theories, and strategies, and how they are related. We also discuss the major goals which have guided the design of the TAME strategies.

TAME Templates. TAME currently provides templates for each of the three classes of automata mentioned in Section 1: LV timed automata, I/O automata, and SCR automata. Each template provides a standard structure for defining an automaton. Because LV timed automata are essentially I/O automata with time added, the template originally designed for specifying LV timed automata was easily adapted to specifying I/O automata. To define an automaton of either of these two classes, the user provides the information indicated in Figure 1.

Template Part	User Fills In	Remarks
`actions`	Declarations of non-time-passage actions	The nondefault cases of the `actions` datatype
`MMTstates`	Type of the "basic state" representing the state variables	Usually a record type
`OKstate?`	An arbitrary state predicate restricting the set of states	Default is `true`
`enabled_specific`	Preconditions for all the non-time-passage actions	`enabled_specific(a) =` specific precondition of action a
`trans`	Effects of all the actions	`trans(a,s)` = state reached from state s by action a
`start`	State predicate defining the initial states	Preferred forms: `s = ...` or `s = (# basic:=basic(s) WITH ... ... #)`
`const_facts`	Predicate describing relations assumed among the constants	Optional

Figure 1: Information in the TAME template

Specifications of I/O automata in the style used by Devillers et al. [12] and Romijn [28] (see Appendix A for an example) can be easily translated into TAME specifications. The definitions of the actions of the I/O automaton provide the names and argument types needed for their TAME declarations, preconditions and effects. The definitions of the state variables and their types in the I/O automaton specification provide the information needed to define the type of the basic state as well as any needed auxiliary type definitions in the TAME specification. The initial state information for the I/O automaton is translated into the initial state predicate `start` of the TAME specification. Finally, any constants and predicates relating constants defined for the I/O automaton can be represented in the TAME specifica-

tion using constant declarations and the axiom `const_facts`. TAME does not provide automated support for composing automata or reasoning directly about an automaton defined as a composition. However, when an I/O automaton is defined as the composition of two or more other I/O automata (this happens with some of the automata in [28]), the user can combine the information extracted from the individual automaton descriptions to produce a single TAME specification in a (usually) straightforward way.

TAME Proof Steps. The standard strategies of TAME are designed to support mechanical reasoning about automata using proof steps that mimic human proof steps. These strategies are based on type and name conventions enforced by the templates, the TAME standard theories, and additional special definitions, auxiliary local theories, and local strategies that can be generated from a particular template instantiation. For example, lemmas in the standard theory `machine` support the induction strategy **AUTO_IN-DUCT** (see Figure 2). The auxiliary local theories contain lemmas used to support rewriting and forward chaining needed in "obvious" reasoning about the particular application.

Reference [2] lists the TAME user strategies useful for I/O and LV automata, and describes their effects. These strategies implement proof steps typically used in hand proofs of automaton properties. Hand proofs of invariant properties typically contain only proof steps from a limited set. Figure 2 lists the most common proof steps used in invariant proofs and names the TAME strategies that implement them. TAME strategies also exist for several steps needed less frequently than those listed in Figure 2.

Proof Step	TAME Strategy	Remarks
Break down into base case and induction (i.e., action) cases, and do standard first steps of each case	AUTO_INDUCT	For starting an induction proof
Appeal to precondition of an action	APPLY_SPECIFIC_PRECOND	Used when needed in induction cases
Apply the inductive hypothesis to argument(s) other than the (default) skolem constant(s)	APPLY_IND_HYP	Used when needed in induction cases; needs argument(s)
Perform the standard initial steps in the direct proof of an invariant	DIRECT_PROOF	For starting a non-induction proof
Apply an auxiliary invariant lemma	APPLY_INV_LEMMA	Used in any proof; needs argument(s)
Break down into cases based on a predicate	SUPPOSE	Used in any proof; needs boolean argument
Apply "obvious" reasoning, e.g., propositional, equational, datatype	TRY_SIMP	Used for "it is now obvious" in any proof
Use a fact from the mathematical theory for a state variable type	APPLY_LEMMA	Used in any proof; needs argument(s)

Figure 2: Common steps for invariant proofs and their TAME strategies[1]

Major Goals of the TAME Proof Steps. One major goal of the TAME proof steps is to save the user from much of the tedium typical of proofs done directly in PVS. One technique for achieving this (used in almost all of the TAME strategies) is to incorporate repeated patterns of steps. Several repeated patterns are incorporated into the TAME strategy **AUTO_INDUCT**. In some cases, a repeated pattern becomes a single proof step. For example, the TAME strategy called **APPLY_INV_LEMMA**, with the appropriate arguments, introduces the desired invariant lemma, instantiates it, expands the invariant, and dis-

[1]Here and below, a name in bold capital letters denotes a TAME strategy.

charges the reachability condition that is the hypothesis of the lemma. Another technique TAME uses to eliminate tedium is to automate certain inferences which are "obvious" to humans but which, in PVS, require detailed user guidance together with knowledge of the behavior and some of the more obscure proof steps of the PVS prover. Several such inferences relate to the PVS DATATYPE construct. For example, if `con` and `des` are a corresponding constructor-destructor pair in a datatype `A`, it is obvious to a human that `con(des(a)) = a` whenever `a` is a "con" value of `A`. To establish this needed fact in a proof, the PVS user must apply the PVS step APPLY-EXTENSIONALITY. Establishing other simple facts about data types can require the PVS steps REPLACE and CASE to do explicit substitution and judicious case splitting. The auxiliary local theories that can be generated from a template instantiation provide the conditional rewrite rules and lemmas used by the TAME strategy **TRY_SIMP** to make such inferences automatic.

A second major goal of the TAME proof steps is to make saved PVS proofs understandable by humans without executing them in PVS. Saved TAME proofs have a clear structure, with the meanings of the proof branches indicated by comments automatically generated by TAME. The meanings of the individual TAME proof steps can be inferred from their names and arguments. In verbose mode, TAME prints extended comments showing the exact facts introduced, so that the reader of a proof does not need to look up particular lemmas, preconditions, etc. (Sections 3 and 5 contain example TAME proofs.)

Finally, the TAME proof steps are designed to give users improved feedback from PVS in the course of a proof, through appropriate labels on formulae that indicate their origin, and hence their significance. For example, the strategy **AUTO_INDUCT** labels formulae in subgoals corresponding to induction cases according to whether they come from the precondition, inductive hypothesis, or inductive conclusion, or express the fact that the prestate or poststate of an induction step is reachable. When the TAME step **SUPPOSE** is applied to an assertion argument P, the current subgoal will be split into two subgoals, one with the hypothesis P labeled `Suppose` and the other with the hypothesis `not(P)` labeled `Suppose not`.[2] Facts introduced with **APPLY_INV_LEMMA** or **APPLY_LEMMA** will be given a label `lemma <lemma-name>` derived from the name of the lemma. In later subgoals, the descendants of a labeled formula retain the labels of their parent and are thus useful in indicating the reason for the presence of new formulae. The ability to label formulae, a recent new feature of PVS, is important in the execution of, as well as the feedback from, the TAME strategies.

3. FIRST CASE STUDY

Reference [12] describes the direct use of PVS to mechanize the proofs of properties of an automaton called *TIP*, a tree identify protocol. Below, we contrast the TAME approach with the direct PVS approach, drawing from the third author's results from applying of TAME to *TIP*. For brevity,

[2]Technically, because PVS does not allow negative formulae to appear at the top level of a sequent, the hypothesis "not(P)" in the "`Suppose not`" subgoal will appear as "P" in the "wrong half" of the sequent (assuming P itself is not a negation).

_____

To prove: For every reachable state `s`, for every edge `e`,
            `length(mq(e,s)) <= 1.`

Proof: The proof is by induction. The assertion is trivial for the base case, i.e., when `s` is an initial state. For each action of *TIP*, it remains to prove the corresponding induction case, i.e., that the assertion is preserved by that action. Only three of the induction cases are nontrivial.

Case 1: The action `add_child(addE)`. Let the value of the parameter `addE` of `add_child` be `addE_action`, and consider the edge `e_theorem`. First, apply the specific precondition of `add_child(addE_action)`. Then, consider separately the cases `e_theorem = addE_action` and `e_theorem ≠ addE_action`. In each of these cases, the proof is now obvious.

Case 2: The action `children_known(childV)`. Let the value of the parameter `childV` of `children_known` be `childV_action`, and consider the edge `e_theorem`. Consider separately two cases. Suppose first that `source(e_theorem) = childV_action`. In this case, first appeal to the specific precondition of `children_known(childV_action)` and then apply Invariant I_2 in the prestate to `e_theorem`. The remainder of the proof in this case is now obvious. The proof in the case `source(e_theorem) ≠ childV_action` is obvious.

Case 3: The action `ack(ackE)`. Let the value of the parameter `ackE` of `ack` be `ackE_action`, and consider the edge `e_theorem`. Consider separately two cases. Suppose first that `e_theorem = ackE_action`. Appeal to the specific precondition of `ack(ackE_action)`. The proof in this case is now obvious. The proof in the case `e_theorem ≠ ackE_action` is obvious.

_____

Figure 3: Natural Language Proof of Invariant I_5

we refer to "TAME" specifications and proofs and "PVS" specifications and proofs.

Comparing Proofs of Invariants. The most dramatic difference between the PVS approach of [12] and the TAME approach is in the proofs of invariants. The TAME proofs are much shorter, and the significance of proof branches and individual proof steps is much clearer. Moreover, the TAME proofs correspond in a very clear way to the hand proofs in [11]. In fact, the TAME proofs for those *TIP* invariants for which hand proofs were provided were done by referring to the hand proofs. (See Section 5 for more details.) This method differs from the method used by the authors of [12], who found the hand proofs of little use in guiding their mechanization in PVS, and did not try to follow them.

Proofs in general have a natural tree structure. Branching occurs when the proof breaks into cases or when extra proof obligations are created by a proof step. When a user creates a proof interactively in PVS, PVS saves an executable script of the proof, recording both the proof steps invoked by the user and the branching structure of the proof. In the example TAME and PVS proofs in this paper, the proof steps supplied by the user are in Roman font with the names of TAME strategies in bold, and the parts of the proof scripts created by PVS are in italics. The italic numbers in quotes represent the addresses of the proof branches in the tree and hence show the tree structure. The TAME proofs also include comments (in italics and preceded by semicolons) automatically generated by the TAME strategies.

The resemblance of TAME proofs to hand proofs is illustrated by the natural language proof of *TIP* Invariant I_5 in Figure 3. This proof was obtained by hand translating the TAME proof of I_5 in Figure 4 in a straightforward way. Although the hand proof from which the TAME proof of I_5 was derived was a Lamport-style proof [18] rather than a natural language proof, TAME proofs can also be (and have been) derived from natural language proofs providing the level of detail of the proof in Figure 3.

Figure 5 presents the PVS proof of *TIP* Invariant I_5 developed by Devillers et al. [12]. As the translation in Fig-

_____

`Inv_5(s:states): bool = (FORALL (e:Edges): length(mq(e,s)) <= 1);`

```
(""
  (AUTO_INDUCT)
  (("1" ;;Case add_child(addE_action)
        (APPLY_SPECIFIC_PRECOND)
        (SUPPOSE "e_theorem = addE_action")
        (("1.1" ;;Suppose e_theorem = addE_action
              (TRY_SIMP))
         ("1.2" ;;Suppose not [e_theorem = addE_action]
              (TRY_SIMP))))
   ("2" ;;Case children_known(childV_action)
        (SUPPOSE "source(e_theorem) = childV_action")
        (("2.1" ;;Suppose source(e_theorem) = childV_action
              (APPLY_SPECIFIC_PRECOND)
→             (APPLY_INV_LEMMA "2" "e_theorem")
              (TRY_SIMP))
         ("2.2" ;;Suppose not [source(e_theorem) = childV_action]
              (TRY_SIMP))))
   ("3" ;;Case ack(ackE_action)
        (SUPPOSE "e_theorem = ackE_action")
        (("3.1" ;;Suppose e_theorem = ackE_action
              (APPLY_SPECIFIC_PRECOND)
              (TRY_SIMP))
         ("3.2" ;;Suppose not [e_theorem = ackE_action]
              (TRY_SIMP))))))
```
_____

Figure 4: TAME Proof of Invariant I_5

_____

`INV_5((s: states)): bool = (FORALL (e: E): length(mq(s)(e)) <= 1)`

```
(""
 (EXPAND "invariant?")          ("2.1.1.3" (ASSERT))))    (ASSERT)
 (PROP)                         ("2.1.2" (INST?))))        (INST?)
 (("1"                          ("2.2"                     (("2.3.1.1.1"
  (SKOSIMP*)                     (SKOSIMP*)                  (EXPAND "member")
  (EXPAND "INV_5")               (EXPAND "steps")            (EXPAND "froms")
  (SKOLEM!)                      (EXPAND "ACK_step")         (ASSERT)
  (EXPAND "Init")                (PROP)                      (HIDE −2 −3 −4)
  (PROP)                         (REPLACE −1 :HIDE? T)       (INST?)
  (HIDE −1)                      (EXPAND "INV_5")            (EXPAND "append")
  (INST?)                        (SKOSIMP*)                  (EXPAND "length" 1)
  (PROP)                         (LIFT-IF)                   (EXPAND "length")
  (HIDE 1)                       (PROP)                      (ASSERT))
  (EXPAND "length")             (("2.2.1"                   ("2.3.1.1.2"
  (ASSERT))                      (INST?)                     (EXPAND "tos")
 ("2"                            (ASSERT)                    (EXPAND "target")
  (SKOLEM 1 (S _ T))             (HIDE −1 2 3)               (EXPAND "inv")
  (INDUCT "a" 1)                 (EXPAND "tl")               (EXPAND "member")
 (("2.1"                         (EXPAND "length")           (EXPAND "froms")
  (SKOSIMP*)                     (LIFT-IF)                    (PROPAX))))
  (EXPAND "steps")               (PROP)                     ("2.3.1.2"
  (EXPAND "A_C_step")           (("2.2.1.1" (ASSERT))        (EXPAND "tos")
  (PROP)                         ("2.2.1.2" (ASSERT)         (EXPAND "member")
  (REPLACE −2 :HIDE? T)          (EXPAND "length")           (EXPAND "froms")
  (EXPAND "INV_5")               (LIFT-IF)                   (EXPAND "inv")
  (SKOSIMP*)                     (ASSERT))))                 (EXPAND "target")
  (LIFT-IF)                     ("2.2.2" (INST?))))          (PROPAX))))
  (PROP)                        ("2.3"                      ("2.3.2" (HIDE −2) (INST?))))
 (("2.1.1"                       (SKOSIMP*)                 ("2.4" (SKOSIMP*)
  (INST?)                        (EXPAND "steps")            (EXPAND "steps")
  (REPLACE −1 :HIDE? T)          (EXPAND "C_K_step")         (EXPAND "R_C_step")
  (HIDE −1)                      (PROP)                      (PROP)
  (HIDE 2)                       (REPLACE −3 :HIDE? T)       (REPLACE −3 :HIDE? T)
  (EXPAND "tl")                  (EXPAND "INV_5")            (EXPAND "INV_5")
  (EXPAND "length")              (SKOSIMP*)                  (PROPAX))
  (ASSERT)                       (LIFT-IF)                  ("2.5"
  (LIFT-IF)                      (PROP)                      (SKOSIMP*)
  (PROP)                        (("2.3.1"                    (EXPAND "steps")
  (ASSERT)                       (INST?)                     (EXPAND "ROOT_step")
  (EXPAND "length")             (("2.3.1.1"                  (PROP)
  (LIFT-IF)                       (ASSERT)                   (REPLACE −2 :HIDE? T)
  (PROP)                       → (USE "INV_2_reach")         (EXPAND "INV_5")
 (("2.1.1.1" (ASSERT))         → (EXPAND "INV_2")            (PROPAX))))))
  ("2.1.1.2" (ASSERT))         → (INST?)
```
_____

Figure 5: PVS Proof of Invariant I_5 (Devillers et al.)

ure 3 of the TAME proof of Invariant I_5 illustrates, a TAME proof can be understood by referring only to the specification of the automaton and its invariants, without rerunning it in the PVS proof checker. PVS proofs in general do not have this property. For example, one must step through the proof in Figure 5 with the PVS proof checker to determine the contributions of many of the steps, such as (PROP), (SKOSIMP*), (HIDE -1), (INST?), (REPLACE -2 :HIDE? T), and (LIFT-IF) in the first column.

The PVS encoding of state invariant lemmas is slightly different from the TAME encoding. Most invariants—those proved by induction—have two associated lemmas: the first lemma states that the invariant holds in start states and is preserved by transitions, and the second (proved trivially from the first) states that the invariant holds for all reachable states. When induction is not required in the proof—i.e., when the invariant follows from other invariants—only the second lemma is needed. The TAME encoding of every state invariant lemma is equivalent to the second PVS lemma. For proofs requiring induction, the strategy **AUTO_INDUCT** first reduces this lemma to the equivalent of the first PVS lemma before performing many of the standard initial proof steps. Thus, the TAME proofs by induction of invariants correspond to the PVS proofs of the first associated lemma in the PVS encoding.

The difference between corresponding TAME proofs and PVS proofs is illustrated by the TAME and PVS proofs for I_5 in Figures 4 and 5. Both proofs were created by interactive use of the PVS prover, with the user supplying all the information in the proofs (except the parts in italics). An obvious difference between the proofs is the number of proof steps: the TAME proof requires the user to supply only 14 steps, while the PVS proof requires the user to supply 111 steps. The two proofs also have different structures. The PVS proof tree (see Figure 5) has two branches at the top level, with the second divided into five parts. The first branch corresponds to the base case of the induction proof, while the five parts into which the second branch divides correspond to the five induction cases—one for each of the five actions of *TIP* (see Appendix A). By contrast, the TAME proof tree (see Figure 4) has three top level branches.

Although the two proofs have different structures, some relationships can be found. The PVS proof clearly has repeating patterns; the TAME strategies take advantage of such repeating patterns to produce higher-level proof steps. One pattern, which appears only once in this proof, is the application of another invariant. The three arrows at the bottom of the second column of the PVS proof in Figure 5 mark the steps that apply Invariant I_2 to the current state (before the action) and the skolem constant for the edge e of Invariant I_5. In the TAME proof in Figure 4, this is accomplished by the proof step (**APPLY_INV_LEMMA** "2" "e_theorem") in the second proof branch (also marked with an arrow). Most of the repeating patterns are handled by either **AUTO_INDUCT** or **TRY_SIMP**. For example, the base case and last two induction cases, whose proofs are "obvious" to a human, are done automatically by **AUTO_INDUCT**. Hence the three top level branches in TAME proof correspond to the branches *"2.1"*, *"2.2"*, and *"2.3"* of the PVS proof (though not in that order) representing the three nontrivial action cases.

The proof execution times in the *TIP* example average about three times as long for the TAME proofs as for their corresponding PVS proofs (e.g., the longest proof—of three invariants combined—took 37 seconds for TAME vs 15 seconds for PVS[3]). However, the relative simplicity and clarity of the TAME proofs strongly suggests that the human time needed to construct the proofs with TAME is considerably shorter than that needed to construct proofs with the PVS-based approach of [12].

Comparing Specifications. As expected of two independent encodings of a problem, the PVS and TAME specifications have rather different structures. The PVS specification of the automaton *TIP* involves a large set of automaton-specific theories with a complex import structure having several (around nine) levels. Moreover, the organization of the import structure is at least partly problem-specific. Thus, how one would use the same methodology to specify a different I/O automaton in PVS is not completely clear. In contrast, the TAME specification of *TIP* is essentially a single automaton-specific theory that imports instantiations of a small collection of generic theories. Only one of these generic theories—the theory states, which combines the non-default part of the state from the TAME template with the default part associated with time values—involves the automaton definition; the others are used in theorem proving support. Hence unlike the PVS specification, the TAME specification of the automaton involves almost no layering of definitions. As a result, the TAME specification is more easily grasped as a whole, and its correspondence to the original I/O automaton description is easier to see. There are additional, automaton-specific theories associated with the TAME specification that can be derived in a standard, automatable way from the automaton specification. These theories supply lemmas to the generic TAME strategies.

The PVS and TAME specifications of *TIP* also differ in the way they capture the transitions. In the PVS specification, each transition is described using the combined information from the precondition and effect of each action. In TAME, the preconditions and effects of actions are defined separately. In a few instances, some information from the precondition is needed as a guard in the definition of the effect for the definition to pass typechecking. Experience with many examples has shown that in practice, this rarely happens. When possible, separating the precondition and effect of an action provides an advantage in creating understandable induction proofs: it allows one to determine just when the precondition is important in the induction step corresponding to that action. Thus, it allows one to determine whether a specification property might be affected when a precondition is changed in the specification.

Beyond Invariants: Simulation and Refinement. Because PVS lacks support for defining a general automaton type and for passing theory parameters to theories, completely general definitions of simulation and refinement (see [22]) are impossible to express in PVS. For this reason, TAME does not yet include specialized support for proving simulations or refinements. However, the PVS specification of *TIP* does include a definition of the refinement relation, using the most convenient general form that can currently

---

[3]These times are for PVS 2.2 on an UltraSPARC-II.

be provided with PVS.[4] In addition, the PVS proofs for *TIP* include a proof that *TIP* is a refinement of another automaton called *SPEC*. In this respect, the PVS specification and proofs have an advantage over the TAME specification and proofs. The generic theories supporting the definition of refinement in the PVS specification can almost certainly be adapted for use with a new TAME "refinement" template. Rather than use this approach, however, a future version of TAME will use PVS support for theory parameters (to be provided in a future version of PVS [19]) to provide a generic refinement template and associated proof strategies.

4. SECOND CASE STUDY

In the second case study, the third author applied TAME first to examples from Romijn's solution to the RPC-Memory Problem [28, 27], and then to *TIP* and its invariants [12, 11]. For the RPC-Memory example, this required specifying three I/O automata (called *Memory**, *MemoryImp*, and *Imp*), and proving 24 associated invariants. In the case of *TIP*, she used TAME to specify the single automaton and to check the proofs of the 15 invariants for which hand proofs were supplied. (Two additional *TIP* invariants for which no hand proofs were given were later proved by the first author using TAME.) Below, we first describe the *TIP* and RPC-Memory examples. We then discuss the time required by the third author, special problems she had to solve, the extent to which the TAME strategies were sufficient for mechanizing the proofs, and some errors in specifications and proofs that were discovered during the mechanization.

The Examples. The *TIP* example from [12] is a specification and analysis of the leader election algorithm forming the core of the tree identify phase of the physical layer of the IEEE 1394 high performance serial multimedia bus protocol. The goal of the analysis is to establish the property "For an arbitrary tree topology, exactly one leader is elected." Half of this property ("at most one leader is elected") is proved as *TIP* Invariant I_{15} (see Appendix A). The RPC-Memory problem, which was posed by Broy and Lamport at the 1994 Dagstuhl Seminar on Reactive Systems, concerns the specification of a memory component and a remote procedure call (RPC) component for a distributed system and the implementation of both. (The TAME specifications and proofs for these examples can be found at the URL http://chacs.nrl.navy.mil/projects/tame.)

Time Required. The third author required approximately a week to read and understand earlier TAME specifications. These specifications include auxiliary theories, which are derived from a template instantiation and currently must be generated by hand. In addition, she needed about a day to learn how to use TAME to obtain a proof.

Once these initial barriers were overcome, specifying *Memory** in TAME and creating its auxiliary theories required about two days, and the proofs of its three invariants, plus a fourth auxiliary invariant, required only a few hours. Some of this time was used to discover the need for and the formulation of the auxiliary invariant. Specifying *MemoryImp* in TAME required only a few days. Proving its 12 invariants required about two weeks. The time required to prove these

12 invariants was longer for a combination of reasons. First, the proofs of these invariants were more complex than the proofs of the *Memory** invariants. Because the proof of one invariant was only loosely sketched, some time was required to determine all of the facts, including an additional invariant lemma, required in its proof. Trying to understand the scopes of the quantifiers in one of the invariants led to a weaker initial formulation of the invariant that was insufficient for the proofs of later invariants, and this had to be rectified. Finally, the third author encountered some situations in which the TAME strategies had to be supplemented by special knowledge and the direct use of PVS. Once appropriate improvements were made to TAME (see Section 5.3), she was able to complete the proofs. After her experience with *Memory** and *MemoryImp*, specifying *Imp* and proving its seven invariants took only three days, and specifying *TIP* and proving its 15 invariants took only five days.

Special Problems. While translating I/O automata specifications into the TAME template is largely straightforward, in this study, creativity was needed for some aspects of the translations. For the most part, these aspects concerned type definitions. For example, the specification of *MemoryImp* required the composition of several I/O automata specifications into a single TAME specification. In such compositions, output actions of one automaton are combined with input actions of another. For at least one combined action, creativity was required in defining the parameter types to make an output action and an input action compatible. In addition, several state variable and action parameter types in the RPC-Memory automata had complex subtype relationships. The third author's original definitions of these relationships in TAME led to several unprovable TCCs (type correctness conditions generated by the PVS typechecker). One approach to making the TCCs provable is to include axioms describing the subtype relationships in the specification. Instead, the third author defined the types as appropriate subtypes of a PVS datatype. Doing this permits the TCCs to be proved automatically in PVS and avoids the possible introduction of inconsistent axioms.

Another case in which some sophistication was needed to represent an I/O automaton in TAME was in a step of *Imp* using a `for` construct to simultaneously update a set of variables whose indices satisfied a certain predicate. Because these variables could not be enumerated, representing this step in TAME required the use of the LAMBDA construct in PVS.

In addition to the information required in the TAME template, auxiliary information is sometimes needed. For the RPC-Memory automata, a few auxiliary functions and predicates defined in the original I/O automata specifications were also included in the TAME specifications. For *TIP*, a few auxiliary results about data structures were also required. A small set of lemmas about the relationship between edges and their reverse edges was needed to mechanize those steps in hand proofs whose justification in [11] was "math". These were simple enough to prove using GRIND, PVS's general automatic proof step.[5] In addition, a subset

---

[4]This definition makes use of a parameterized automaton type defined in a theory parameterized by the action and state types.

[5]GRIND fails to terminate on one of the proofs and needs to be helped by APPLY-EXTENSIONALITY in another. Evidence exists that these complications are due to a bug in PVS. We have reported the bug to the PVS implementors.

of the theory of tree-structured digraphs was needed in the proof of Invariant I_{15}. Rather than using the full theory developed by Devillers et al., the third author simply determined the essential fact needed about such digraphs—that they are connected—and included it as an axiom. Using this axiom, she proved several auxiliary invariants needed in the proof of I_{15}.

Sufficiency of the TAME Strategies. Once improvements to the TAME strategies (due to feedback from the third author) were complete, the strategies listed in Figure 2 were almost sufficient to obtain all of the proofs for the RPC-Memory example. **APPLY_IND_HYP** and **APPLY_LEMMA** were not needed. In a few places, new TAME strategies (**INST_IN** and **SKOLEM_IN**, which are described in Section 5.3) and the PVS steps EXPAND and INST were used.

The proofs of the *TIP* invariants used all of the strategies in Figure 2, together with **INST_IN**, **SKOLEM_IN**, EXPAND, and INST; in addition, the PVS step SPLIT was used to separate threads in the combined proofs of several lemmas, and two additional TAME steps, **COMPUTE_POSTSTATE** and **DIRECT_INDUCTION** were required. The step **COMPUTE_POSTSTATE** was needed to introduce facts about the poststate required in a proof in which it was natural to refer to the poststate in a supposition introduced with **SUPPOSE**. The proof of one auxiliary invariant lemma for *TIP* introduced by the third author required induction over the natural numbers, but not over the reachable states of *TIP*. She mechanized this proof using variants of the PVS SKOLEM command, PVS's EXPAND, INDUCT, and INST commands, and TAME's **APPLY_INV_LEMMA** strategy. The step **DIRECT_INDUCTION** was developed to prove invariants whose proof requires mathematical induction followed by direct (non-induction) proofs of the branches. With the aid of **DIRECT_INDUCTION**, the proof can be mechanized using PVS's EXPAND, together with the TAME strategies **APPLY_INV_LEMMA**, **SKOLEM_IN**, and **APPLY_IND_HYP** (used to apply the mathematical induction hypothesis). Whether **DIRECT_INDUCTION** will be useful in other examples is an open question.

Specification and Proof Errors Discovered. The specifications and proofs of both the RPC-Memory and *TIP* examples were very carefully done. Thus, the third author uncovered only a few errors. Specifying the RPC-Memory automata in TAME and applying the PVS typechecker exposed a few cases of incompleteness and inconsistency in the specifications. For example, the intended types of certain constants were unclear, and there was a type inconsistency in the definition and use of one function. No specification errors were found in the *TIP* example, which is not surprising, since this example had already been proved in PVS. But the PVS proofs for *TIP* were not derived from the hand proofs, so although the *TIP* invariants had been checked, their hand proofs had not been checked. Using TAME, the third author discovered a few cases of incorrect inferences or justifications in both the hand proofs for *TIP* and the RPC-Memory proofs. She was able to correct all of these problems in the TAME proofs, usually in a very straightforward way, and in one case by identifying and proving an auxiliary invariant. Thus, like Rudnicki and Trybulec [29],

she found that Lamport-style proofs, though very structured and detailed, are still informal and as a result may contain incorrect or missing details. Her results led to corrections by Romijn and Devillers et al. in both the specifications and proofs in [28, 27, 11].

5. DISCUSSION

This section presents several observations resulting from our case studies. First, as indicated in Sections 3 and 4, TAME proofs are readily constructed from hand proofs that give sufficient detail. A hand proof that indicates which facts were used on which proof branches and which subcases need to be considered usually provides sufficient detail; details of inferences drawn from the facts are normally not required. In previous applications (e.g. [3, 5]), the hand proofs mechanized with TAME were English language proofs. As stated in Section 4, the majority of the proofs mechanized by the third author using TAME were Lamport-style proofs. These proved to be as straightforward to mechanize in TAME as English language proofs. Section 5.1 gives an example of the correspondence between a Lamport-style proof and the TAME proof derived from it. Second, as noted in Sections 2 and 3, TAME proofs are intended to be understandable without reference to the PVS proof checker. In Section 5.2, we describe how TAME proofs can actually be interpreted as English language proofs, using the TAME proof from Section 5.1 as an example. Finally, several improvements were made to TAME as a result of the third author's experience. Section 5.3 discusses these improvements and some issues they raise.

5.1 Constructing TAME Proofs

The proof in Figure 6 (provided to the reader as an aid) describes in English the complete TAME proof (see Figure 7) of *TIP* Invariant I_4. As an illustration of how a TAME proof can be obtained from a Lamport-style proof, Figures 7 and 8 show the correspondence between the TAME steps and the relevant steps from a Lamport-style proof of Invariant I_4. These relevant steps are shown in Figure 8, which contains the single branch of the hand proof that TAME found to be nontrivial; the remainder of the Lamport-style proof was done automatically by TAME.

To understand the relationship between the two kinds of proofs, we can compare the Lamport-style and TAME proofs of I_4 in Figures 8 and 7. In the Lamport-style proof in Figure 8, the values s and t represent the prestate and poststate in the induction step, and the values f, g, and v' are, respectively, the skolem constants for the quantified variables e, f, and v in I_4, which TAME automatically names `e_theorem`, `f_theorem`, and `v_theorem`. The action $C\_KNOWN(v)$ in the Lamport proof corresponds to the action `children_known(childV_action)` in the TAME proof; the name `childV_action` is constructed automatically by TAME from the name of the formal parameter `childV` of `children_known`. We added annotations (see the right-hand column of Figure 7) to the TAME proof to show, for each of its steps or branches, the step of the Lamport proof containing a corresponding inference or justification. For example, the appeal "by IH" to the inductive hypothesis at step $< 3.1 >$ in the Lamport-style proof is handled automatically by TAME's **AUTO_INDUCT** strategy since, for this proof, the correct instantiation of its variables is the

---

To prove: For every reachable state `s`,
        for all edges `e` and `f` and every vertex `v`,
        if `target(e) = target(f) = v` and `e ≠ f`,
        then `init(v)` or `child(e)` or `child(f)`.

Proof: The proof is by induction. The assertion is trivial for the base case, i.e., when `s` is an initial state. For each action of *TIP*, it remains to prove the corresponding induction case, i.e., that the assertion is preserved by that action. Only one of the induction cases is nontrivial: the case of the action `children_known(childV)`.

Thus, consider the case when the action is `children_known(childV)`. Let the value of the parameter `childV` of `children_known` be `childV_action`. Let the state before this action be `prestate`. Let the edges `e` and `f` and the vertex `v` be `e_theorem`, `f_theorem`, and `v_theorem`. Consider separately two cases. Suppose first that `v_theorem = childV_action`. In this case, introduce the fact that

    `init(childV_action, prestate)` ∧

  ∀ `e:tov(childV_action)`, `f:tov(childV_action)`.

      `child(e,prestate)` ∨ `child(f,prestate)` ∨ `e = f`

by appealing to the specific precondition of `children_known(childV_action)` in the state `prestate`. Instantiating the second part of this precondition with `e_theorem` and `f_theorem`, the remainder of the proof in this case is now obvious. The fact that `e_theorem` and `f_theorem` belong to the subtype `tov(childV_action)` of the type `Edges` is also obvious. The proof in the case `v_theorem ≠ childV_action` is obvious.

Figure 6: English translation of TAME proof of I_4

Inv_4(s:states): bool =
    (FORALL (e,f:Edges, v:Vertices):
                (target(e)=v and target(f)=v and not(e=f))
                      => (init(v,s) or child(e,s) or child(f,s)))

("" (**AUTO_INDUCT**)	
;; *Case children_known(childV_action)*	< 3 >, < 3.1 >, < 3.2 >,
	< 3.2.1 >
(**SUPPOSE** "v_theorem = childV_action")	< 3.2.2 >
(("1" ;; *Suppose v_theorem = childV_action*	< 3.2.3 >
(**APPLY_SPECIFIC_PRECOND**)	< 3.2.3.1 >
;; *Applying the precondition*	
;; *init(childV_action, prestate)*	
;; *AND*	
;; *(FORALL (e: tov(childV_action)):*	
;; *FORALL (f: tov(childV_action)):*	
;; *child(e, prestate) OR*	
;; *child(f, prestate) OR e = f)*	
(INST "specific-precondition_part_2"	
"e_theorem" "f_theorem")	
(("1.1" (**TRY_SIMP**))	< 3.2.3.2 >, < 3.2.3.3 >
("1.2" (**TRY_SIMP**))	< 3.2.3.1 >
("1.3" (**TRY_SIMP**))))	< 3.2.3.1 >
("2" ;; *Suppose not [v_theorem = childV_action]*	< 3.2.4 >
(**TRY_SIMP**))))	< 3.2.4.1 >, < 3.2.4.2 >
	< 3.2.5 >, < 3.3 >

Figure 7: Complete TAME proof (verbose) of I_4

$I_4 = \forall_{e,f,v} target(e) = target(f) = v \wedge e \neq f \Rightarrow init(v) \vee child[e] \vee child[f]$

<3>	Assume $a = C\_KNOWN(v)$, $v \in \mathbf{V}$	
<3.1>	. $s \models I_4$	(by IH)
<3.2>	. Take arbitrary f, g, v' such that	
	$target(f) = target(g) = v \wedge f \neq g$	
<3.2.1>	. $s \models init(v') \vee child[f] \vee child[g]$	
<3.2.2>	. Case distinction on $v' = v$	
<3.2.3>	. . Assume $v' = v$	
<3.2.3.1>	. . . $s \models child[f] \vee child[g]$	(pre. $C\_KNOWN(v)$ and $f, g \in to(v)$)
<3.2.3.2>	. . . $t \models child[f] \vee child[g]$	(eff. $C\_KNOWN(v)$ does not change *child*)
<3.2.3.3>	. . . $t \models init(v) \vee child[f] \vee child[g]$	
<3.2.4>	. . Assume $\neg(v' = v)$	
<3.2.4.1>	. . . $s \models init(v') \vee child[f] \vee child[g]$	(by <3.2.1>)
<3.2.4.2>	. . . $t \models init(v') \vee child[f] \vee child[g]$	(eff. $C\_KNOWN(v)$ does not change *child*
		or $init[v']$ by <3.2.4>)
<3.2.5>	. . $t \models init(v') \vee child[f] \vee child[g]$	
<3.3>	. $t \models I_4$	(def. I_4)

Figure 8: Nontrivial branch of Lamport-style proof of I_4

---

skolem constants. The only steps the TAME user must supply, besides **TRY_SIMP**, are the **SUPPOSE** for the case distinction at step < 3.2.2 > and the **APPLY_SPECIFIC_PRECOND** and INST corresponding to application of the precondition to f and g at step < 3.2.3.1 >. Checking that f and g are of type $to(v)$ is handled by proving the TCCs generated by PVS as the result of the instantiation step INST—this is accomplished by the proof steps **TRY_SIMP** at *"1.2"* and *"1.3"* in the TAME proof. Introducing the effect of the action and setting up Invariant I_4 in the poststate as a proof goal are both handled automatically in the TAME proof by **AUTO_INDUCT**, and appeals to previous proof steps are handled automatically in the TAME proof by the final **TRY_SIMP**.

5.2 Explaining TAME Proofs

Because the meanings of TAME proof steps are essentially independent of the proof state current at the time they are executed by the PVS proof checker, TAME proofs can be understood from their saved scripts by referring to the original specification and by knowing the conventions TAME uses for skolemization and instantiation. Thus, given the TAME proof of Figure 7, it is fairly straightforward to derive the equivalent English language proof in Figure 6. Knowledge of TAME's conventions about skolemization is used in specializing the action parameter `childV` to `childV_action`, and s, e, f, and v in the theorem to `prestate`, `e_theorem`, `f_theorem`, and `v_theorem`. One convention about instantiation is used in the INST command in the TAME proof: a precondition in the form of a conjunction is broken down into "_part_1", "_part_2", and so on, in order. A second convention about instantiation is reflected in the English translation in Figure 3 of the **APPLY_INV_LEMMA** step in the TAME proof in Figure 4: unless a state argument is given, the invariant lemma is applied to the state `prestate`. One additional question that arises in interpreting the TAME proof in Figure 7 is why the INST step in the proof results in three subgoals instead of one. When extra subgoals from an INST occur, PVS has generated one or more TCCs as extra proof branches, requiring the user to show that values used in the instantiation have the correct types.

Aside from the problem with possible TCCs, the derivation of an English language proof from a TAME proof is straightforward enough to be automatable, and in fact, we have recently implemented a prototype translator of saved TAME proofs. Note that the TAME proof in Figure 7 is a verbose TAME proof, in contrast to the non-verbose TAME proof in Figure 4. Thus, details such as the actual fact introduced by **APPLY_SPECIFIC_PRECOND** in Figure 7 can be incorporated into the English version. Had the proof in Figure 4 been verbose, the actual fact introduced by **APPLY_INV_LEMMA** would have been displayed in the TAME proof (as well as the facts introduced by each of the three uses of **APPLY_SPECIFIC_PRECOND**). An alternative to translating TAME proofs from their saved scripts to obtain an English language version is to create an English language version simultaneously with the TAME version. Implementing this technique would allow even more detail to be incorporated in the English version, if desired, and would better facilitate interpreting extra TCC subgoals in English.

5.3 Improvements Made to TAME

Several improvements were made to the template, the strategies, and the supporting theories of TAME as the result of feedback from the third author. The first improvement generalizes the template. Improvements to the strategies have made TAME more user-friendly by reducing the amount of low-level reasoning associated with certain proof steps. Improvements to the supporting theories extend the scope of the high-level reasoning supported in TAME. We discuss these improvements below, along with some issues they raise.

Improving the Template. The base case of induction proofs corresponding to the start states is usually trivial to prove. The strategy **AUTO_INDUCT** is designed to prove the base case automatically, when possible. Although none of the TAME templates enforce any condition on the form of the start state predicate `start`, the automatic proof of the base case by AUTO_INDUCT works best if `start(s)` is expressed as an equality `s = <start-state>`, where `<start-state>` is a record value. In previous applications of TAME, each automaton had a single start state, and thus the convention was that `<start-state>` was an explicit record giving the initial values of all state variables.

In the RPC-Memory example, the third author encountered an automaton in which the start state was not unique: initial values were given for only some of the basic state variables. She therefore developed a new template convention for the start state, in which `<start-state>` is a record with its time-related components assigned the standard initial values and its `basic` component (representing the basic state variables) assigned its "old" value updated with values for those variables whose initial values are specified. This is easily done using the PVS construct WITH for updating records and functions and has the effect of leaving the non-updated variables of the basic state unspecified, as desired. Moreover, the strategy **AUTO_INDUCT** works just as well for proving base cases with the new conventional form for `start(s)` as it did with the old one.

Improving the Strategies. As noted in Section 4, the third author had difficulty translating a few of the steps from hand proofs into TAME. One such step was the application of an invariant lemma to the poststate of a transition in an induction step. The default used by the TAME step **APPLY_INV_LEMMA** is to apply the lemma to `prestate`. TAME previously represented the poststate as `trans(<action>,prestate)`, where `<action>` is the action of the induction step, and maintained among the hypotheses the fact that `trans(<action>,prestate)` is reachable, to facilitate application of invariant lemmas to the poststate. However, this representation of the poststate complicated applying an invariant lemma. Not only did the user have to supply `trans(<action>,prestate)` as an argument to **APPLY_INV_LEMMA**, where `<action>` itself could be an expression with parameters, but after doing this, the user had to explicitly expand the transition function `trans`. The third author's difficulties inspired improvements to **AUTO_INDUCT** and **APPLY_INV_LEMMA** that hide this complexity from the user. The term `trans(<action>,prestate)` is now represented simply as `poststate`, and to apply an invariant lemma to the post-

*	**1.**	length(emptylist) = 0
	2.	$\forall$ L:list. length(L) = 0 $\Rightarrow$ L = emptylist
	3.	$\forall$ n:nat, e:element, L:list. length(L) = n $\Rightarrow$ length(cons(e,L)) = n+1
*	**4.**	$\forall$ e:element, L:list. L = emptylist $\Rightarrow$ length(cons(e,L)) = 1
*	**5.**	$\forall$ n:nat, e:element, L:list. L $\neq$ emptylist $\wedge$ length(L) = n $\Rightarrow$ length(cdr(L)) = n$-$1
*	**6.**	$\forall$ n:nat, e:element, L:list. L $\neq$ emptylist $\wedge$ length(L) $\leq$ n $\Rightarrow$ length(cdr(L)) $\leq$ n$-$1
	7.	$\forall$ L:list. (length(L) < 0) = FALSE

Figure 9: Rewrite rules for lists used by TAME

state, the user simply applies **APPLY_INV_LEMMA** to the argument `poststate` and any other arguments to the lemma.

As noted in reference [4], the inability of the user to instantiate or skolemize with respect to embedded quantifiers in PVS sometimes makes it difficult to follow the structure of a hand proof using PVS. The third author encountered this problem in some proofs of the RPC-Memory example. To address the problem, two new strategies, called **INST_IN** and **SKOLEM_IN**, were added to TAME to approximate internal instantiation and skolemization. These strategies perform automated simplification in an attempt to handle the non-quantified parts of a formula, and then use the standard PVS proof steps INST and SKOLEM. Although some wasteful proof branching can result (this happened with one RPC-Memory lemma), this approach handles embedded quantifiers well in many cases.

Improving the Supporting Theories. The state variables used in I/O automata specifications do not always have simple types. For example, some automata from the RPC-Memory example use state variables that must be represented as "datatypes" using the PVS DATATYPE construct. As noted in Section 2, TAME supports "obvious" reasoning about datatypes using auxiliary theories generated from a template instantiation. Previous to the third author's use of TAME, these auxiliary theories contained only lemmas to support reasoning about the datatype `actions`.[6] Because of the additional datatypes used in the RPC-Memory automata, the auxiliary theories now include lemmas from *all* datatypes in a template instantiation.

In the specification of the automaton *TIP* from [12] (see Appendix A), the type of the state variable `mq(e)`, where `e` is an edge, is defined to be `Bool*`, that is, a list of Booleans. Several of the *TIP* invariant lemmas involving `mq(e)` require reasoning about lengths of lists. Because PVS has a built-in type `list[T]`, where T is a type parameter, it is reasonable to add auxiliary lemmas to support reasoning about lengths of lists. Figure 9 shows the set of lemmas used as rewrite rules for lists in TAME. Those rules with an asterisk were actually applied by TAME in proving the *TIP* invariants. The PVS proof in Figure 5 contains several instances of the PVS command '(EXPAND "length")'; these mark places where simple reasoning about length is occurring. With the rules in Figure 9, the TAME user does not need to guide PVS through this reasoning.

While one might argue that rewrite rules for lists should be standard in TAME because `list[T]` is a standard type in

---

[6] For LV timed automata, there are also lemmas for the datatype `time`.

PVS, there are many examples in which state variables have other complex types, as we discovered in applying TAME to some of the invariant lemmas of [14]. For reasons of proof efficiency, the number of rewrite rules that are always active should be limited to those that are relevant. Thus, a practical approach for handling "obvious" reasoning about complex types is to use a library of PVS theories containing the lemmas needed to support such reasoning. Such theories need to be developed with care; we do not guarantee the theory in Figure 9 to be the best theory to support obvious reasoning about lengths of lists. The extent to which existing libraries developed by other members of the PVS user community would be useful in TAME is still to be determined.

6. RELATED WORK

An increasing number of proof assistants, including assistants for the Duration Calculus [30], for the TRIO logic [1], and for proving invariant properties of DisCo specifications [17], use PVS as the underlying prover. The Duration Calculus and TRIO assistants support proofs using steps from particular logics. The DisCo assistant supports proofs of properties of DisCo specifications, using Lamport's Temporal Logic of Actions, with specialized PVS strategies generated by a compiler. These strategies, though uniform in concept, are specific to each given application. A similar approach was used in an earlier version of TAME; PVS enhancements, especially the documentation of the internal structure of PVS sequents, have allowed us to make the TAME strategies more generic.

Several researchers have applied mechanical theorem provers to LV timed automata or I/O automata. In addition to the application of PVS described in [12], reference [20] describes how the Larch theorem prover LP was used to prove properties of several protocols specified as LV timed automata, and reference [26] describes a verification environment for I/O automata based on Isabelle; like [12], both include simulation proofs as well as proofs of invariants. In addition, [26] develops a detailed metatheory for I/O automata. TAME has an advantage over Larch and Isabelle: it produces compact, informative proof scripts. Although Larch provides detailed proof scripts with some information on the content of a proof, Larch does not support the matching of high level natural proof steps with user-defined strategies, nor the automatic documentation of a proof through comments provided by TAME. While Isabelle tactics perform some of the services of the TAME strategies [26], Isabelle does not save proof scripts for completed proofs.

A toolset has been developed that provides an automatic translator from the IOA language for I/O automata to Larch specifications and an interface to the Larch theorem prover LP [15]. This toolset will eventually include a similar translator to PVS that is being developed by Devillers and Vaandrager; a prototype now exists [13]. TAME currently has a prototype translator from specifications in the SCR language to TAME specifications [6], and an automatic translator from IOA specifications is planned.

7. CONCLUSION

In [24], Miller discusses several major problems encountered in the AAMP5 project, in which PVS was used to prove the correctness of a set of microcode instructions. Two problems were how to organize the specification, and how to structure complex proofs. He also notes that the learning curve in this project was very steep, that many supporting theories had to be developed, and that the robustness of proofs became a concern when specifications were modified.

Within its domain of application, TAME solves most of these problems. In particular, it provides templates to organize specifications of automata, high level proof steps designed to make proof structures more understandable, and supporting theories appropriate to the domain. The third author's experience with TAME lends support to our belief that the learning curve for TAME is much less steep than that for the direct use of PVS. TAME proofs tend to be fairly robust, because they use high level proof steps that do not depend on details of the sequent in the current proof goal. (This dependence is present in several places in the PVS proof in our first case study.) In addition, TAME proofs are usually easy to modify when a change in a specification requires some changes in a proof.

In [24], Miller also notes that productivity in the AAMP5 project required the same individuals to serve as both domain experts and PVS experts. Because TAME proofs can be understood separately from PVS, we believe that TAME can provide a way to allow domain experts to understand the results of a verification without becoming PVS experts, and to communicate high-level proof outlines to PVS (or TAME) experts that can be easily checked in TAME. This has been demonstrated to some extent by our previous experience with TAME and by the third author's experience with the RPC-Memory example.

Thus, we believe that specialized interfaces such as TAME can solve many of the problems associated with introducing the use of PVS into industrial practice. This is consistent with the point of view of Crow and Di Vito [10], who state in that:

> Applying formal methods "right out of the box" is difficult. Tailoring the methods to the application at hand is both necessary and desirable.

As noted in Section 2, TAME is based on template specifications for given system models, standard supporting theories, and special strategies to implement reasoning steps appropriate to the models. With an appropriate automatic specification translator, a specialized interface can also allow developers to create specifications in an environment familiar to them. For TAME, such a translator has been developed for specifications created in the SCR toolset [6]. We believe that the same methods can be followed to create specialized interfaces in other application domains. In fact, similar methods were used to some extent in the other PVS-based proof assistants discussed in Section 6. An open question is whether specialized interfaces such as TAME can be developed to address the needs of practitioners working in different application domains.

8. REFERENCES

[1] Andrea Alborghetti, Angelo Gargantini, and Angelo Morzenti. Providing automated support to deductive analysis of time critical systems. In *Proc. 6th European Software Engineering Conference (ESEC/FSE'97)*, Lect. Notes in Comp. Sci., pages 211–226. Springer-Verlag, 1997.

[2] M. Archer. Tools for simplifying proofs of properties of timed automata: The TAME template, theories, and strategies. Technical Report NRL/MR/5540–99-8359, NRL, Wash., DC, 1999.

[3] M. Archer and C. Heitmeyer. Mechanical verification of timed automata: A case study. In *Proc. 1996 IEEE Real-Time Technology and Applications Symp. (RTAS'96)*, pages 192–203. IEEE Computer Society Press, 1996.

[4] Myla Archer and Constance Heitmeyer. Human-style theorem proving using PVS. In E. L. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics (TPHOLs'97)*, volume 1275 of *Lect. Notes in Comp. Sci.*, pages 33–48. Springer-Verlag, 1997.

[5] Myla Archer and Constance Heitmeyer. Verifying hybrid systems modeled as timed automata: A case study. In *Hybrid and Real-Time Systems (HART'97)*, volume 1201 of *Lect. Notes in Comp. Sci.*, pages 171–185. Springer-Verlag, 1997.

[6] Myla Archer, Constance Heitmeyer, and Steve Sims. TAME: A PVS interface to simplify proofs for automata models. In *Proc. User Interfaces for Theorem Provers 1998 (UITP '98)*, Eindhoven, Netherlands, July 1998.

[7] Ricky W. Butler. An introduction to requirements capture using PVS: Specification of a simple autopilot, NASA Technical Memorandum 110255. NASA Langley Research Center, May 1996.

[8] Ricky W. Butler, James L. Caldwell, Victor A. Carreño, C. Michael Holloway, Paul S. Miner, and Ben L. Di Vito. NASA Langley's research and technology-transfer program in formal methods. In *Proc. 10th Annual Conf. on Computer Assurance (COMPASS'95)*, pages 135–149, Gaithersburg, MD, June 1995. IEEE Computer Society Press.

[9] Judith Crow and Ben L. Di Vito. Formalizing space shuttle software requirements. In *Proc. First ACM Workshop on Formal Methods in Software Practice (FMSP'96)*, pages 40–48, San Diego, CA, January 1996.

[10] Judith Crow and Ben L. Di Vito. Formalizing space shuttle software requirements: Four case studies. *ACM Transactions on Software Engineering and Methodology*, 7(3):296–332, July 1998.

[11] M. Devillers. Verification of a tree-identity protocol. URL http://www.cs.kun.nl/~marcod/1394.html, 1997.

[12] M. Devillers, D. Griffioen, J. Romijn, and F. Vaandrager. Verification of a leader election protocol—formal methods applied to IEEE 1394. *Formal Methods in System Design*, 16(3):307–320.

[13] Marco Devillers. Private communication. January, 1999.

[14] A. Fekete, N. Lynch, and A. Shvartsman. Specifying and using a partitionable group communication service. In *Proc. Sixteenth Ann. ACM Symp. on Principles of Distributed Computing (PODC'97)*, pages 53–62, Santa Barbara, CA, August 1997.

[15] S. J. Garland and N. A. Lynch. The IOA Language and Toolset: Support for Designing, Analyzing, and Building Distributed Systems. Draft. MIT Laboratory for Computer Science, August, 1998.

[16] C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Trans. on Softw. Eng.*, 24(11):927–948, November 1998.

[17] Pertti KelloMaki. *Mechanical Verification of Invariant Properties of DisCo Specifications*. PhD thesis, Tampere University of Technology, Finland, November 1997.

[18] L. Lamport. How to write a proof. Technical report, Digital Equipment Corp., System Research Center, February 1993. Research Report 94.

[19] Patrick Lincoln. Private communication. July, 1998.

[20] Victor Luchangco. Using simulation techniques to prove timing properties. Master's thesis, Massachusetts Institute of Technology, June 1995.

[21] N. Lynch and M. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.

[22] N. Lynch and F. Vaandrager. Forward and backward simulations – Part I: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995.

[23] N. Lynch and F. Vaandrager. Forward and backward simulations – Part II: Timing-based systems. *Information and Computation*, 128(1):1–25, July 1996.

[24] Steve Miller. The industrial use of formal methods: Was Darwin right? In *Proc. 2nd IEEE Workshop on Industrial-Strength Formal Specification Techniques*, pages 74–82, Boca Raton, FL, October, 1998.

[25] Steve Miller and Mandayam Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *Proc. Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, FL, April, 1995.

[26] Olaf Mueller. *A Verification Environment for I/O Automata Based on Formalized Meta-Theory*. PhD thesis, Technische Universitaet Muenchen, September 1998.

[27] J. Romijn. Tackling the RPC-Memory Specification Problem with I/O automata. Addendum. URL `http://www.cwi.nl/~judi/papers/` `dagstuhl_proofs.ps.gz`.

[28] J. Romijn. Tackling the RPC-Memory Specification Problem with I/O automata. In M. Broy, S. Merz, and K. Spies, editors, *Formal Systems Specification — The RPC-Memory Specification Case*, volume 1169 of *Lect. Notes in Comp. Sci.*, pages 437–476. Springer-Verlag, 1996.

[29] P. Rudnicki and A. Trybulec. A note on "How to Write a Proof". In *Proc. 1992 Workshop on Types and Proofs for Programs*, June 1996. Available through P. Rudnicki's web page at http://www.cs.ualberta.ca/~piotr/Mizar/.

[30] J. Skakkebaek and N. Shankar. Towards a duration calculus proof assistant in PVS. In *Third Intern. School and Symp. on Formal Techniques in Real Time and Fault Tolerant Systems, Lect. Notes in Comp. Sci. 863.* Springer-Verlag, 1994.

APPENDIX

A. THE I/O AUTOMATON *TIP* FROM [12]

TIP

Internal: *ADD_CHILD, CHILDREN_KNOWN, RESOLVE_CONTENTION, ACK*
Output: *ROOT*

State Variables: $init : \mathbf{V} \to \mathbf{Bool}$
$contention : \mathbf{V} \to \mathbf{Bool}$
$root : \mathbf{V} \to \mathbf{Bool}$
$child : \mathbf{E} \to \mathbf{Bool}$
$mq : \mathbf{E} \to \mathbf{Bool}*$

Init: $\forall v, e : init[v]$
$\wedge \neg contention[v]$
$\wedge \neg root[v]$
$\wedge \neg child[e]$
$\wedge mq[e] = \mathsf{empty}$

Actions:

$ADD\_CHILD(e : \mathbf{E})$
Precondition :
$\wedge init[\mathsf{target}(e)]$
$\wedge mq[e] \neq \mathsf{empty}$
Effect :
$child[e] := 1$
$mq[e] := \mathsf{tl}(mq[e])$

$ACK(e : \mathbf{E})$
Precondition :
$\wedge \neg init[\mathsf{target}(e)]$
$\wedge mq(e) \neq \mathsf{empty}$
Effect :
$contention[\mathsf{target}(e)] := \neg\mathsf{hd}(mq[e])$
$mq[e] := \mathsf{tl}(mq[e])$

$RESOLVE\_CONTENTION(e : \mathbf{E})$
Precondition :
$\wedge contention[\mathsf{source}(e)]$
$\wedge contention[\mathsf{target}(e)]$
Effect :
$child[e] := 1$
$contention[\mathsf{source}(e)] := 0$
$contention[\mathsf{target}(e)] := 0$

$ROOT(v : \mathbf{V})$
Precondition :
$\wedge \neg init[v]$
$\wedge \neg contention[v]$
$\wedge \neg root[v]$
$\wedge \forall e \in \mathsf{to}(v) : child[e]$
Effect :
$root[v] := 1$

$CHILDREN\_KNOWN(v : \mathbf{V})$
Precondition :
$\wedge init[v]$
$\wedge \forall e, f \in \mathsf{to}(v) : child[e] \vee child[f] \vee e = f$
Effect :
$init[v] := 0$
for $e \in \mathsf{from}(v)$ **do** $mq[e] := \mathsf{append}(child[e^{-1}], mq[e])$

TIP Invariant $I_{15} \equiv$
$(\exists v \forall e \in \mathsf{to}(v) : child[e]) \to (\exists! v \forall e \in \mathsf{to}(v) : child[e])$

"There is at most one node for which all incoming links are child links."